

SQL:
Indexes and performance

2 TIN Databases II



INDEXES and PERFORMANCE

Is performance still relevant?

Because:

Transistor density on a manufactured semiconductor doubles about every 18 months.

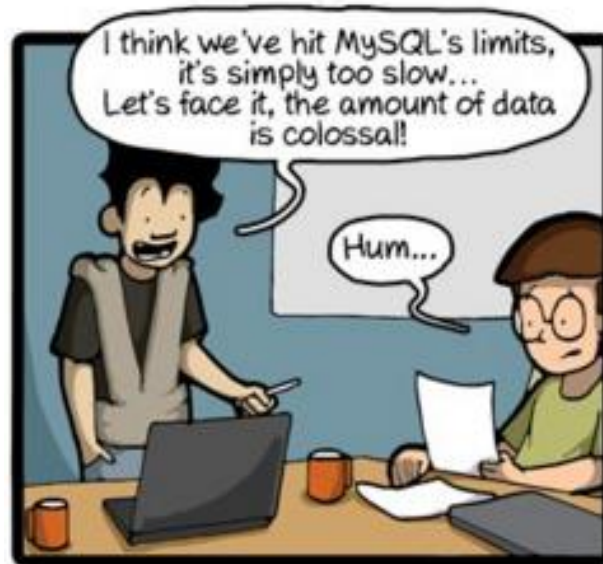
*Moore's law
(no longer valid since 2016?)*

But:

Software gets slower faster than hardware gets faster

Wirth's law

Anyway...



Often indexes offer the solution



CommitStrip.com

Space allocation by SQL Server

- SQL Server uses random access files
- Space allocation in *extents* and *pages*
- Page = 8 kB block of contiguous space
- Extent = 8 logical consecutive pages.
 - uniform extents: for one db object
 - mixed extents: can be shared by 8 db objects (=tables, indexes)
- New table or index: allocation in mixed extent
- Extension > 8 pages: in uniform extent

Database name:

Owner:

☒ Use full-text indexing

Database files:

Logical Name	File Type	Filegroup	Initial Size (MB)
Xtreme_Data	ROWS Data	PRIMARY	14
Xtreme_Log	LOG	Not Applicable	32

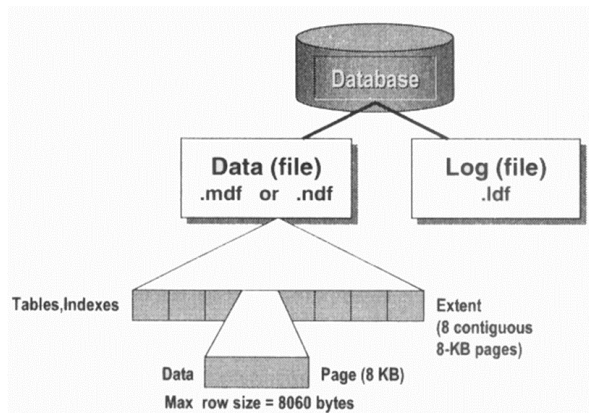


Table scan

- heap: unordered collections of data-pages without clustered index (see below) = default storage of a table.
- access via Index Allocation Map (IAM)
- table scan: if a query fetches all pages of the table → always to avoid!
- Other performance issues with heap:
 - fragmentation: table is scattered over several, non-consecutive pages
 - forward pointers: if a variable length row (e.g. varchar fields) becomes longer upon update, a forward pointer to another page is added.
→ table scan even slower.

Does my query cause a table scan?

Examine the Execution Plan of the query
(db xtreme, with extra script Employeeidx.sql):

The screenshot shows the Microsoft SQL Server Enterprise Manager interface. The 'Query' menu is open, displaying various options. The 'Include Actual Execution Plan' option is highlighted. In the background, the 'Object Explorer' shows the 'Database' folder expanded, with 'Employee2' selected. The 'SQLQuery1.sql' window is open, showing the query: `1 select * from Employee2`. The 'Execution plan' tab is selected, showing the query cost (relative to the batch) as 100%. The execution plan diagram shows a 'SELECT' operation with a cost of 0%, and a 'Table Scan' operation for '[Employee2]' with a cost of 100%.

SQLQuery1.sql - JOHAN-PC\SQL2014.xtreme (Johan-PC\Johan (52))* - Microsoft SQL Server Mana

File Edit View Query Project Debug Tools Window Help

Connection

- Open Server in Object Explorer Alt+F8
- Specify Values for Template Parameters...
- Execute F5
- Cancel Executing Query Alt+Break
- Parse Ctrl+F5
- Display Estimated Execution Plan
- IntelliSense Enabled Ctrl+Q, Ctrl+I
- Trace Query in SQL Server Profiler
- Analyze Query in Database Engine Tuning Advisor
- Design Query in Editor...
- Include Actual Execution Plan**
- Include Client Statistics
- Reset Client Statistics
- SQLCMD Mode
- Results To
- Query Options...

Object Explorer

Connect

JOHAN-PC

Database

System

Adventureworks

Employee2

SQLQuery1.sql - NB1...(EDU\jcor864 (56))*

1 select * from Employee2

121 %

Results Messages Execution plan

Query 1: Query cost (relative to the batch): 100%

select * from Employee2

SELECT Cost: 0 %

Table Scan [Employee2] Cost: 100 %

Compare 2 queries

(db xtreme, with extra script Employeeidx.sql):

- Execute the 2 queries together (select both + Execute!)
- table Employee2 is a copy of Employee, but without indexes
- Query on Employee2 takes 19x longer!

SQLQuery1.sql - NB1...(EDU\jcor864 (56))*

```
1 select lastname from employee order by lastname;
2 select lastname from employee2 order by lastname;
3
```

121 %

Results Messages Execution plan

Query 1: Query cost (relative to the batch): 5%
select lastname from employee order by lastname

SELECT
Cost: 0 %

Index Scan (NonClustered)
[Employee].[EmpLastName]
Cost: 100 %

Query 2: Query cost (relative to the batch): 95%
select lastname from employee2 order by lastname

SELECT
Cost: 0 %

Sort
Cost: 82 %

Table Scan
[Employee2]
Cost: 18 %

What is the difference? Indexes!

- what?
 - ordered structure imposed on records from a table
 - Fast access through tree structure (B-tree=balanced tree)
- why?
 - Can speed up data retrieval
 - Can force unicity of rows
- Why not ?
 - indexes consume storage (overhead)
 - Indexes can slow down updates, deletes and inserts because indexes has to be updated too.

Indexes: library analogy

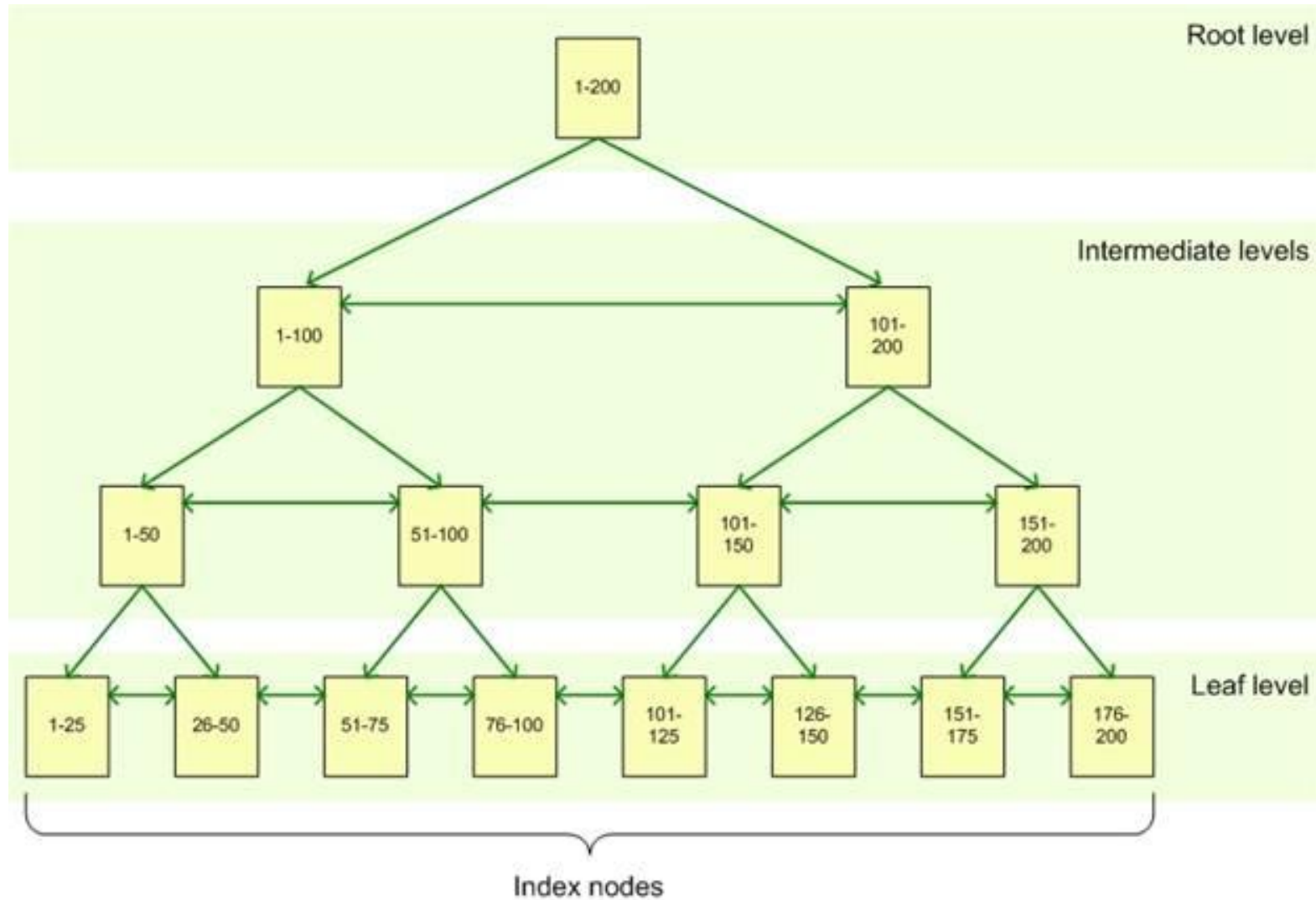
Consider a card catalog in a library. If you wanted to locate a book named Effective SQL, you would go to the catalog and locate the drawer that contains cards for books starting with the letter E (maybe it will actually be labeled D–G). You would then open the drawer and flip through the index cards until you find the card you are looking for. The card says the book is located at 601.389, so you must then locate the section somewhere within the library that houses the 600 class. Arriving there, you have to find the bookshelves holding 600–610. After you have located the correct bookshelves, you have to scan the sections until you get to 601, and then scan the shelves until you find the 601.3XX books before pinpointing the book with 601.389.

In an electronic database system, it is no different. The database engine needs to first access its index on data, locate the index page(s) that contains the letter E, then look within the page to get the pointer back to the data page that contains the sought data. It will jump to the address of the data page and read the data within that page(s). Ergo, an index in a database is just like the catalog in a library. Data pages are just like bookshelves, and the rows are like the books themselves. The drawers in the catalog and the bookshelves represent the B-tree structure for both index and data pages.

SQL Optimizer

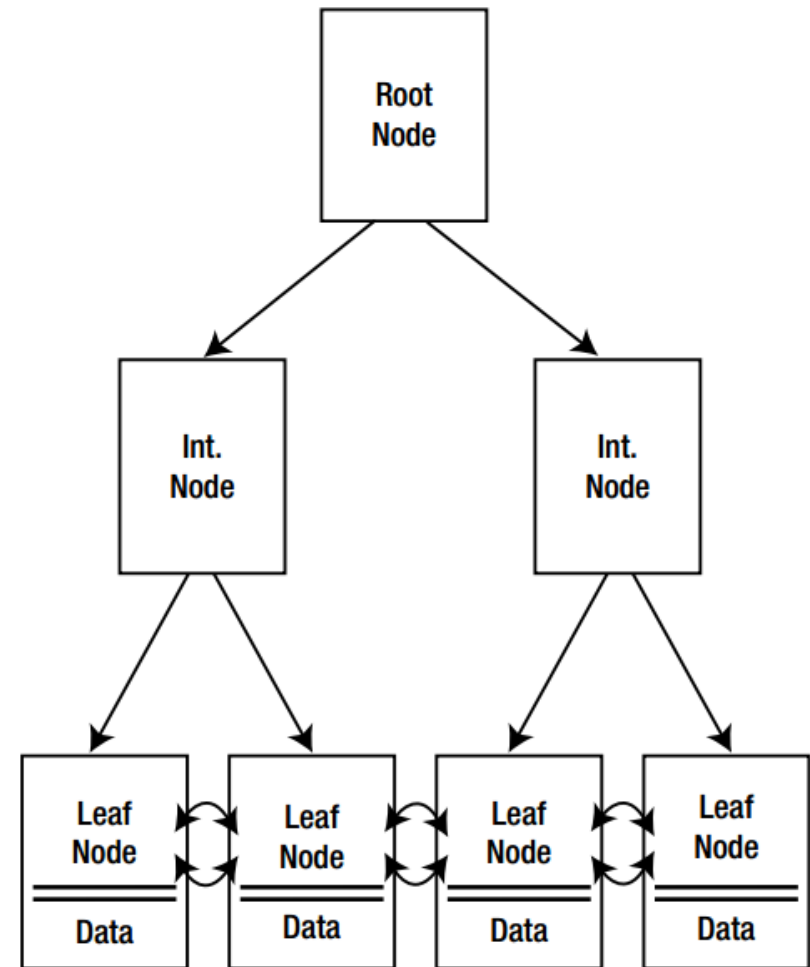
- SQL Optimizer: module in each DBBMS
- Analyses and rephrases each SQL command sent to the DB
- Decides optimum strategy for e.g. index use based on statistics about table size, table use and data distribution.
- In SQL searching is used for fields in *where*, *group by*, *having* and *order by* clauses and for fields that are *joined*.

Indexes as B-trees



Clustered index

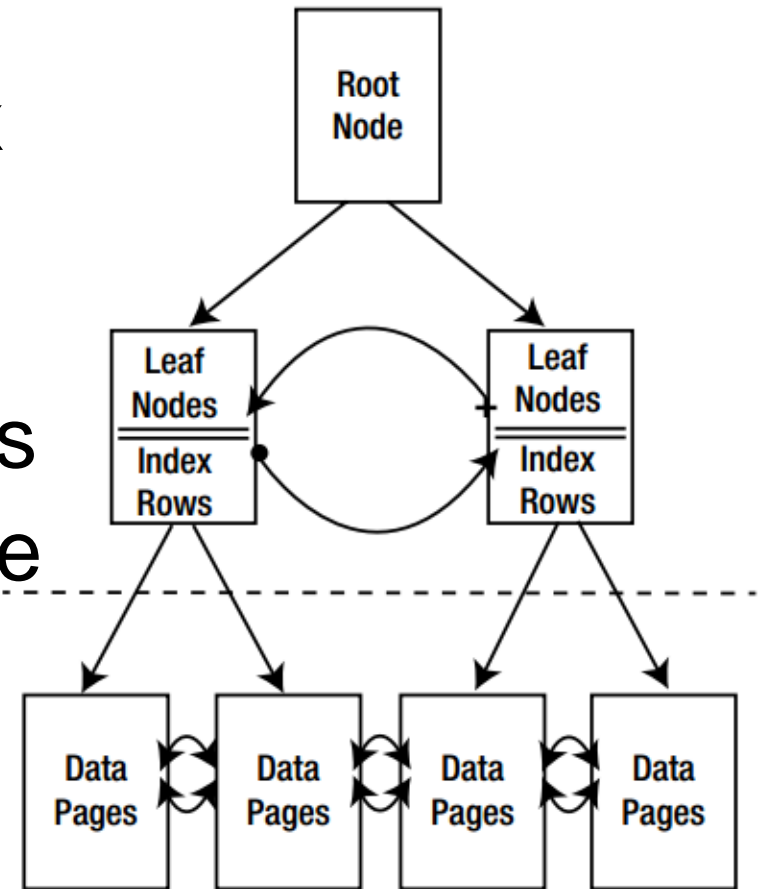
- The physical order of the rows in a table corresponds to the order in the clustered index.
- As a consequence, each table can have only one clustered index.
- The clustered index imposes unique values and the primary keys constraint
- Advantages as opposed to table scan:
 - double linked list ensures order when reading sequential records
 - no forward pointers necessary



Int. Node = intermediate(tussenliggende) node

Non clustered index

- default index
- slower than clustered index
- > 1 per table allowed
- Forward and backward pointers between leaf nodes
- each *leaf* contains key value and *row locator*
 - to position in clustered index if it exists
 - otherwise to heap



Non clustered index

- if query needs more fields than present in index, these fields have to be fetched from data pages.
- when reading via non-clustered index:

either:

- RID lookup = bookmark lookups to the heap using RID's (= row identifiers)

or:

- key lookup = bookmark lookups to a clustered index, if present

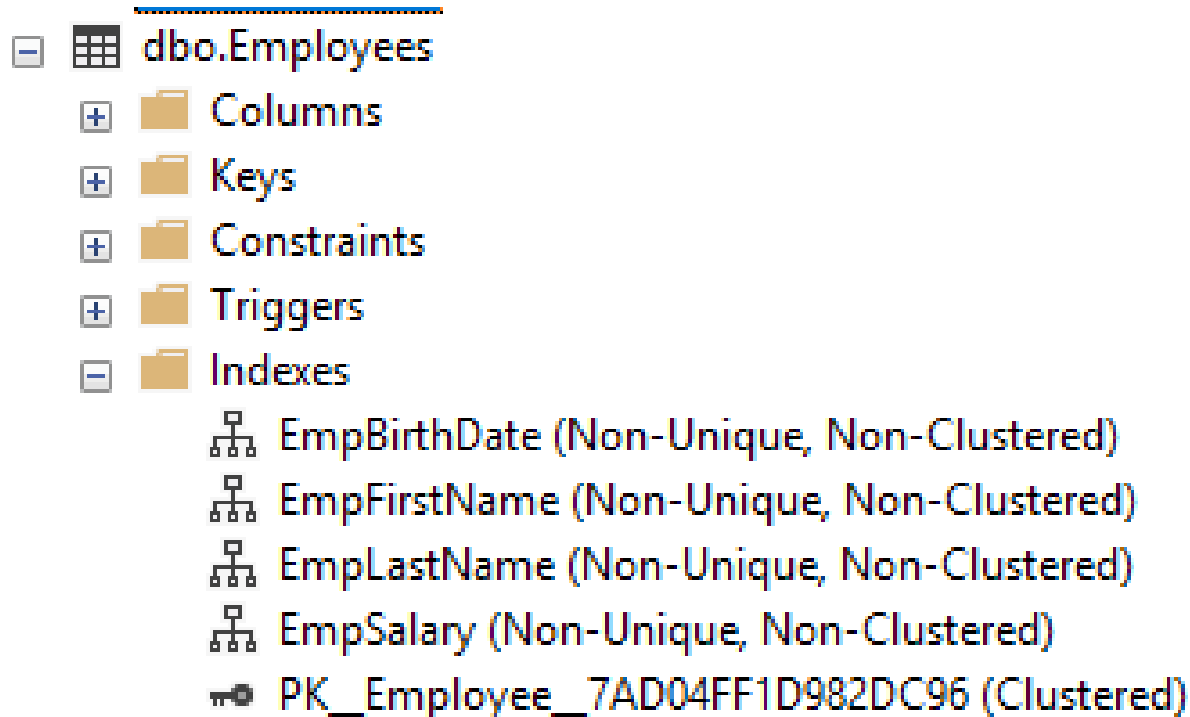
Covering index

- if a non clustered index not completely *covers* a query, SQL Server performs a lookup for each row to fetch the data
- covering index=non-clustered index containing all columns necessary for a certain query
- with SQL Server you can add extra columns to the index (although those columns are not indexed!)

Covering index: example

(db xtreme, with extra script EmployeeIdx.sql):

Current indexes on table Employee:
each index indexes a single field.



Covering index: example

(db xtreme, with extra script EmployeeIdx.sql):

SQLQuery1.sql - NB1...(EDU\jcor864 (56))*

```
1 select lastname from employee where lastname='Duffy';
2
3 select lastname, title from employee where lastname='Duffy';
4
```

121 %

Results Messages Execution plan

Query 1: Query cost (relative to the batch): 33%

SELECT [lastname] FROM [employee] WHERE [lastname]=@1

Index Seek (NonClustered)
[Employee].[EmpLastName]
Cost: 100 %

- index seek via nonclustered index for seeking lastname=N'Duffy'

Query 2: Query cost (relative to the batch): 67%

SELECT [lastname],[title] FROM [employee] WHERE [lastname]=@1

Nested Loops (Inner Join)
Cost: 0 %

Index Seek (NonClustered)
[Employee].[EmpLastName]
Cost: 50 %

Key Lookup (Clustered)
[Employee].[PK_Employee]
Cost: 50 %

- key_lookup up in clustered index (= data) for fetching Title (not in index)

Covering index: example (cont'd)

Solution: covering index via INCLUDE

```
create nonclustered index EmpLastName_Incl_Title  
ON employee(lastname) INCLUDE (title);
```

SQLQuery1.sql - NB1...(EDU\jcor864 (56))*

```
1 select lastname from employee where lastname='Duffy';  
2  
3 select lastname,title from employee where lastname='Duffy';
```

121 %

Results Messages Execution plan

Query 1: Query cost (relative to the batch): 50%
SELECT [lastname] FROM [employee] WHERE [lastname]=@1

SELECT
Cost: 0 %

Index Seek (NonClustered)
[Employee].[EmpLastName_Incl_Title]
Cost: 100 %

Query 2: Query cost (relative to the batch): 50%
SELECT [lastname],[title] FROM [employee] WHERE [lastname]=@1

SELECT
Cost: 0 %

Index Seek (NonClustered)
[Employee].[EmpLastName_Incl_Title]
Cost: 100 %

1 index with several columns vs. several indexes with 1 column

```
create nonclustered index EmpLastName ON employee(lastname);  
+  
create nonclustered index EmpFirstname ON  
employee(firstname);
```

OR

?

```
create nonclustered index EmpLastNameFirstname ON  
employee(lastname,firstname);
```

1 index with several columns vs. several indexes with 1 column

Rule in SQL Server:

When querying (ex. in where-clause) only 2nd and or 3th, ...field of index, it is not used. This directly follows from the B-tree table structure of the composed index

So:

```
SELECT LASTNAME, FIRSTNAME  
FROM EMPLOYEE2  
WHERE FIRSTNAME = 'Chris';
```

does not use the double index

Conclusion: make your indexes according to the most commonly used queries.

1 index met several columns vs. several indexes with 1 column

Test: only combined index on Lastname, Firstname

SQLQuery2.sql - NB1...(EDU\jcor864 (54))* × SQLQuery1.sql - NB1...(EDU\jcor864 (56))*

```
1 SELECT LASTNAME, FIRSTNAME
2 FROM EMPLOYEE2 WHERE LASTNAME = 'Preston';
3
4 SELECT LASTNAME, FIRSTNAME
5 FROM EMPLOYEE2 WHERE FIRSTNAME = 'Chris';
6
```

121 %

Results Messages Execution plan

Query 1: Query cost (relative to the batch): 3%

SELECT [LASTNAME],[FIRSTNAME] FROM [EMPLOYEE2] WHERE [LASTNAME]=@1

SELECT
Cost: 0 %

Index Seek (NonClustered)
[Employee2].[EmpLastNameFirstname]
Cost: 100 %

Query 2: Query cost (relative to the batch): 97%

SELECT [LASTNAME],[FIRSTNAME] FROM [EMPLOYEE2] WHERE [FIRSTNAME]=@1

SELECT
Cost: 0 %

Index Scan (NonClustered)
[Employee2].[EmpLastNameFirstname]
Cost: 100 %

1 index met several columns vs. several indexes with 1 column

test: extra index on Firstname:

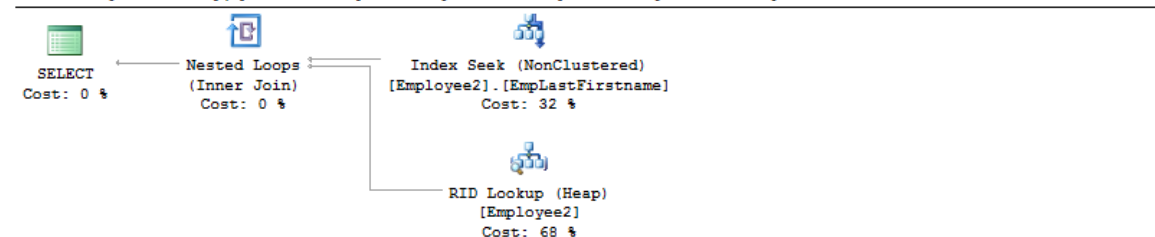
```
create nonclustered index EmpLastFirstname ON employee2(firstname);
```

```
SQLQuery2.sql - NB1...(EDU\jcor864 (54))*  SQLQuery1.sql - NB1...(EDU\jcor864 (56))*
1 SELECT LASTNAME, FIRSTNAME
2 FROM EMPLOYEE2 WHERE LASTNAME = 'Preston';
3
4 SELECT LASTNAME, FIRSTNAME
5 FROM EMPLOYEE2 WHERE FIRSTNAME = 'Chris';
6
```

121 %
Results Messages Execution plan
Query 1: Query cost (relative to the batch): 24%
SELECT [LASTNAME],[FIRSTNAME] FROM [EMPLOYEE2] WHERE [LASTNAME]=@1



Query 2: Query cost (relative to the batch): 76%
SELECT [LASTNAME],[FIRSTNAME] FROM [EMPLOYEE2] WHERE [FIRSTNAME]=@1



not a spectacular improvement because of fetching lastname through lookup

→ covering index with include 'lastname'

1 index met several columns vs. several indexes with 1 column

test: with extra index on Firstname and covering of Lastname

```
create nonclustered index EmpLastFirstnameIncLastname  
ON employee2(firstname) INCLUDE (lastname);
```

```
SQLQuery2.sql - NB1...(EDU\jcor864 (54))* X SQLQuery1.sql - NB1...(EDU\jcor864 (56))*  
1 SELECT LASTNAME, FIRSTNAME  
2 FROM EMPLOYEE2 WHERE LASTNAME = 'Preston';  
3  
4 SELECT LASTNAME, FIRSTNAME  
5 FROM EMPLOYEE2 WHERE FIRSTNAME = 'Chris';  
6
```

now query execution
times are equal.

121 %

Results Messages Execution plan

Query 1: Query cost (relative to the batch): 50%

```
SELECT [LASTNAME],[FIRSTNAME] FROM [EMPLOYEE2] WHERE [LASTNAME]=@1
```

Index Seek (NonClustered)
[Employee2].[EmpLastNameFirstname]
Cost: 100 %

Query 2: Query cost (relative to the batch): 50%

```
SELECT [LASTNAME],[FIRSTNAME] FROM [EMPLOYEE2] WHERE [FIRSTNAME]=@1
```

Index Seek (NonClustered)
[Employee2].[EmpLastFirstnameIncLas...]
Cost: 100 %

Use of indexes with functions and wildcards

SQLQuery2.sql - NB1...(EDU\jcor864 (54))* SQLQuery1.sql - NB1...(EDU\jcor864 (56))*

```
1 SELECT lastname,firstname
2 FROM employee2 WHERE lastname = 'Preston';
3
4 SELECT lastname,firstname
5 FROM employee2 WHERE substring(lastname,2,1) = 'r';
6
7 SELECT lastname,firstname
8 FROM employee2 WHERE lastname like '%r%';
```

121 %

Results Messages Execution plan

Query 1: Query cost (relative to the batch): 2%

SELECT [lastname],[firstname] FROM [employee2] WHERE [lastname]=@1

SELECT
Cost: 0 %

Index Seek (NonClustered)
[Employee2].[EmpLastNameFirstname]
Cost: 100 %

Query 2: Query cost (relative to the batch): 50%

SELECT lastname,firstname FROM employee2 WHERE substring(lastname,2,1) = 'r'

SELECT
Cost: 2 %

Index Scan (NonClustered)
[Employee2].[EmpLastNameFirstnameIncLas...]
Cost: 98 %

Query 3: Query cost (relative to the batch): 49%

SELECT lastname,firstname FROM employee2 WHERE lastname like '%r%'

SELECT
Cost: 0 %

Index Scan (NonClustered)
[Employee2].[EmpLastNameFirstnameIncLas...]
Cost: 100 %

Creation of indexes: syntax

```
CREATE [UNIQUE] [| NONCLUSTERED]  
INDEX index_name ON table (kolom [...n])
```

create index

```
create index ssnr_index on student(ssnr)
```

create index

- **unique** all values in the indexed column should be unique
- remark:
 - when defining an index the table can be empty or filled.
 - columns in a **unique** index should have the **not null** constraint

removing indexes

```
DROP INDEX table_name.index [,...n]
```

deleting index

```
drop index student.SSNR_Index
```

Working with indexes

- SQL Server Management Studio

The screenshot shows the SQL Server Enterprise Explorer tree on the left, with the 'Indexes' folder expanded under the 'dbo.Orders' table. A blue arrow points from the 'SQL Server Management Studio' bullet point to the 'New Index' dialog box. The dialog box is titled 'New Index' and has a 'Ready' status bar. It features a 'Select a page' section with 'General', 'Options', 'Storage', 'Filter', and 'Extended Properties'. The 'Connection' section shows 'JOHAN-PC\SQL2014 [Johan-PC\Johan]'. The 'Table name' field is 'Orders'. The 'Index name' field is 'NonClusteredIndex-20150304-152042'. The 'Index type' is 'Nonclustered'. The 'Unique' checkbox is unchecked. The 'Index key columns' tab is active, showing a table with columns: Name, Sort Order, Data Type, Size, Identity, Allow NULLs, Add..., and Remove. The table contains one row: OrderDate, Ascending, datetime, 8, No, Yes.

Table name: Orders

Index name: NonClusteredIndex-20150304-152042

Index type: Nonclustered

☐ Unique

Index key columns Included columns

Name	Sort Order	Data Type	Size	Identity	Allow NULLs	Add...	Remove
OrderDate	Ascending	datetime	8	No	Yes		

When to use an index

- **which columns should be indexed?**
 - primary and unique columns are index automatically
 - foreign keys often used in joins
 - columns often used in search conditions (WHERE, HAVING, GROUP BY) or in joins
 - columns often used in the ORDER BY clause
- **which columns should not be indexed?**
 - columns that are rarely used in queries
 - columns with a small number of possible values (e.g. gender)
 - columns in small tables
 - columns of type bit, text or image

Tips & tricks

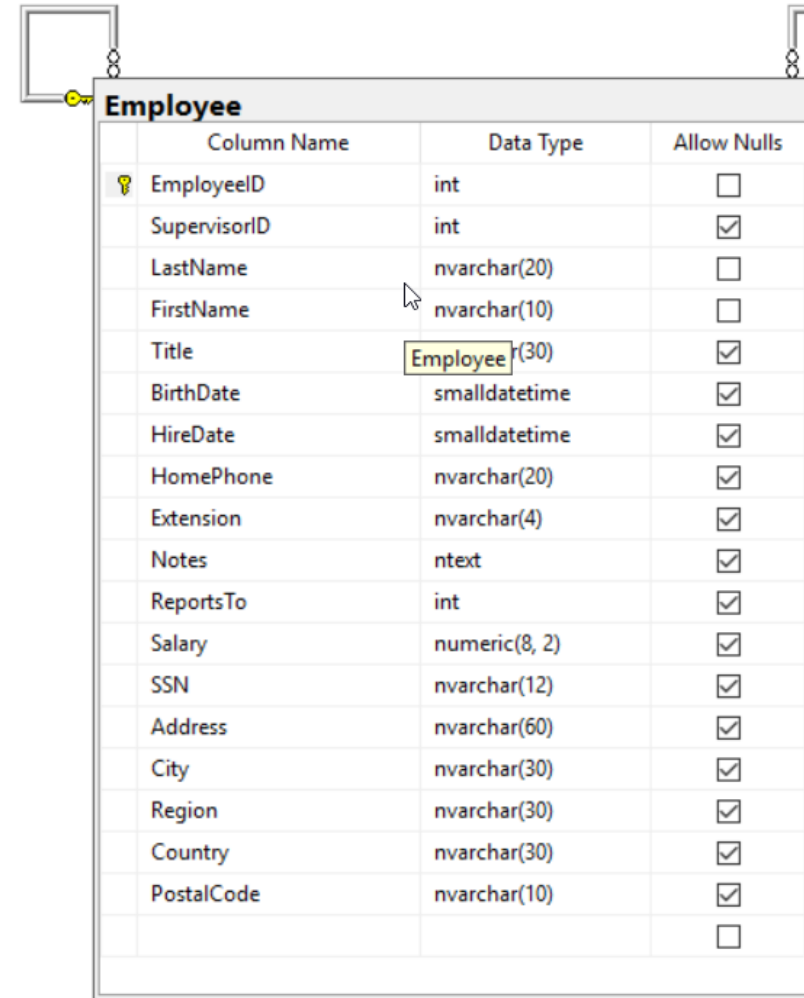
DB xtreme:

```
CREATE INDEX EmpFirstName ON  
Employee (FirstName ASC);
```

```
CREATE INDEX EmpLastName ON  
Employee (LastName ASC);
```

```
CREATE INDEX EmpDOB ON  
Employee (BirthDate ASC);
```

```
CREATE INDEX EmpSalary ON  
Employee (Salary ASC);
```



A screenshot of a database table structure for 'Employee'. The table has 19 columns. The first column, 'EmployeeID', is the primary key. The 'Title' column has a value 'Employee' entered in the data field. The 'Allow Nulls' column indicates which columns can contain null values.

Column Name	Data Type	Allow Nulls
EmployeeID	int	<input type="checkbox"/>
SupervisorID	int	<input checked="" type="checkbox"/>
LastName	nvarchar(20)	<input type="checkbox"/>
FirstName	nvarchar(10)	<input type="checkbox"/>
Title	nvarchar(30)	<input checked="" type="checkbox"/>
BirthDate	smalldatetime	<input checked="" type="checkbox"/>
HireDate	smalldatetime	<input checked="" type="checkbox"/>
HomePhone	nvarchar(20)	<input checked="" type="checkbox"/>
Extension	nvarchar(4)	<input checked="" type="checkbox"/>
Notes	ntext	<input checked="" type="checkbox"/>
ReportsTo	int	<input checked="" type="checkbox"/>
Salary	numeric(8, 2)	<input checked="" type="checkbox"/>
SSN	nvarchar(12)	<input checked="" type="checkbox"/>
Address	nvarchar(60)	<input checked="" type="checkbox"/>
City	nvarchar(30)	<input checked="" type="checkbox"/>
Region	nvarchar(30)	<input checked="" type="checkbox"/>
Country	nvarchar(30)	<input checked="" type="checkbox"/>
PostalCode	nvarchar(10)	<input checked="" type="checkbox"/>
		<input type="checkbox"/>

The following slides provides some general rules of thumb that are applicable in most cases on most databases. They are not carved in stone.

The employee table used in the examples has about 20.000 records.

Tips & tricks: (1) avoid the use of functions

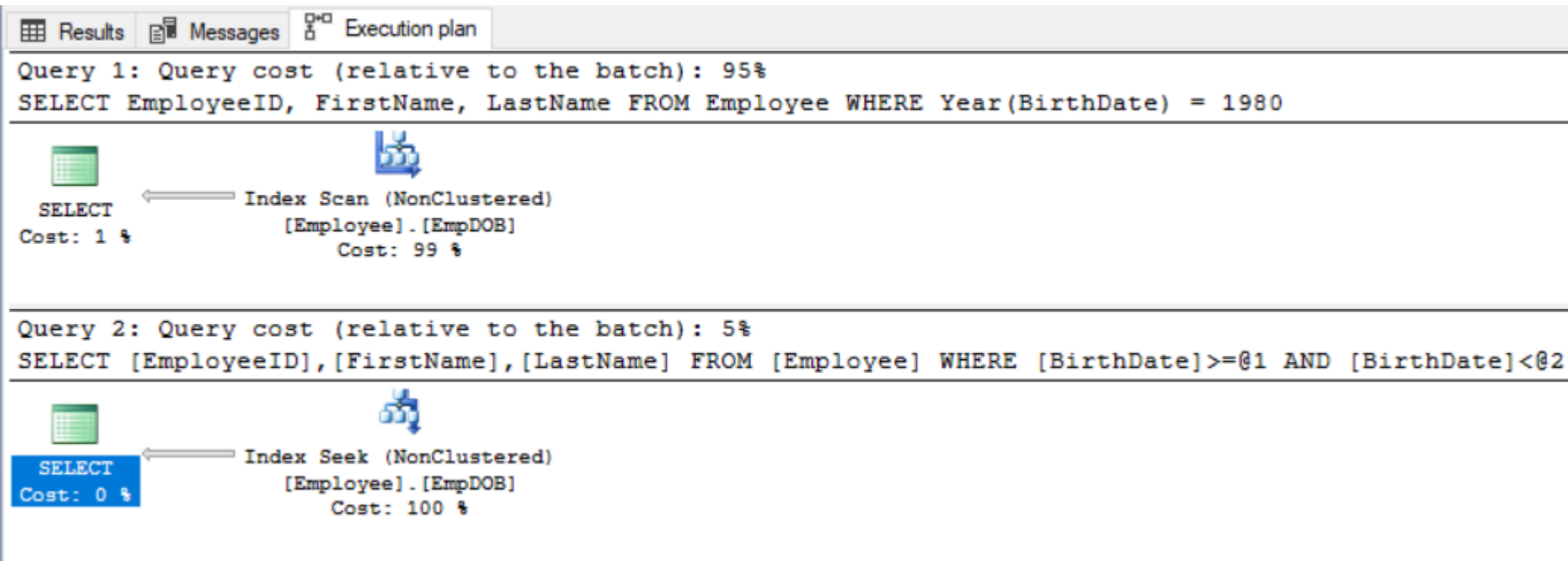
-- BAD

```
SELECT EmployeeID, FirstName, LastName  
FROM Employee  
WHERE Year(BirthDate) = 1980;
```

-- GOOD

```
SELECT EmployeeID, FirstName, LastName  
FROM Employee  
WHERE BirthDate >= '1980-01-01'  
AND BirthDate < '1981-01-01';
```

Tips & tricks: (1) avoid the use of functions



- Index Scan: index is used but it is scanned from the start till the searched records are found.
- Index Seek: tree structure of index is used, resulting in very fast data retrieval.

Tips & tricks: (2) avoid the use of functions

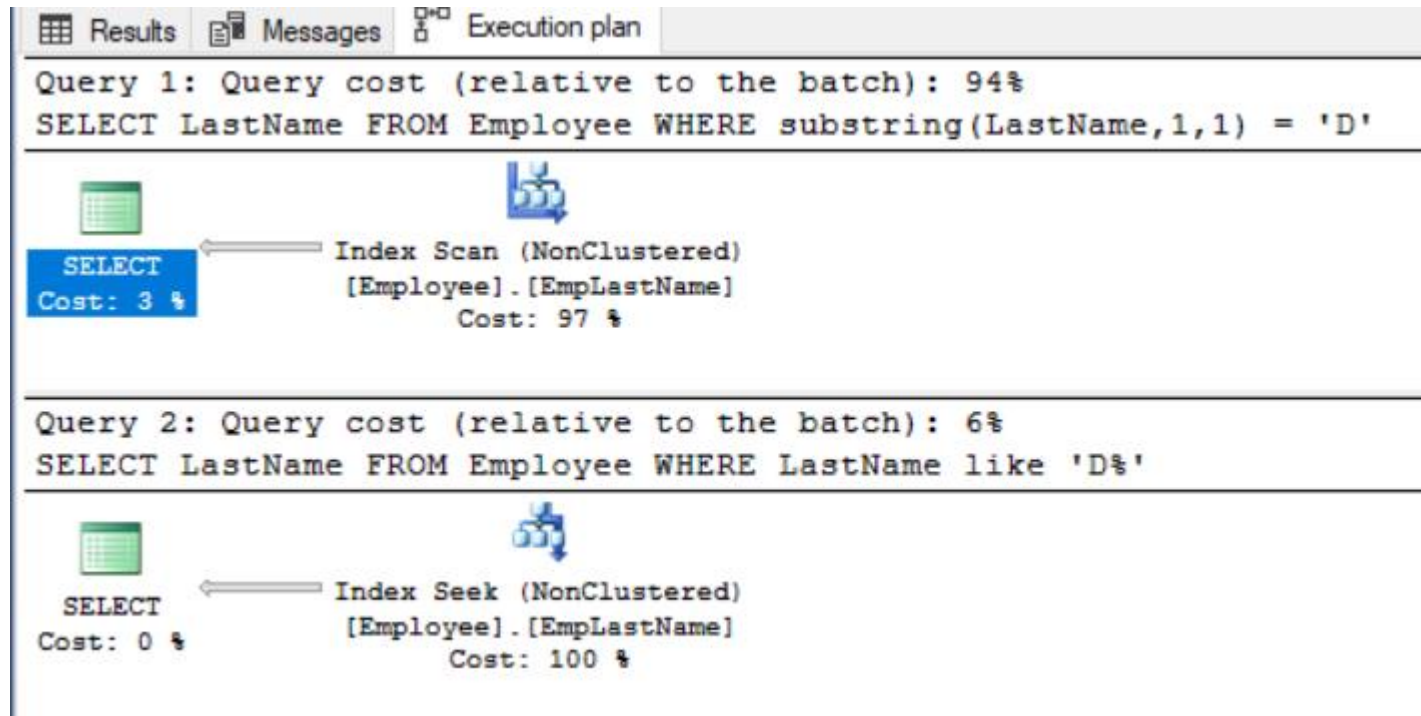
-- BAD

```
SELECT LastName  
FROM Employee  
WHERE substring(LastName,1,1) = 'D';
```

-- GOOD

```
SELECT LastName  
FROM Employee  
WHERE LastName like 'D%';
```

Tips & tricks: (2) avoid the use of functions



- Index Scan: index is used but it is scanned from the start till the searched records are found.
- Index Seek: tree structure of index is used, resulting in very fast data retrieval.

Tips & tricks

(3) avoid calculations, isolate columns

-- BAD

```
SELECT EmployeeID, FirstName, LastName  
FROM Employee  
WHERE Salary*1.10 > 100000;
```

-- GOOD

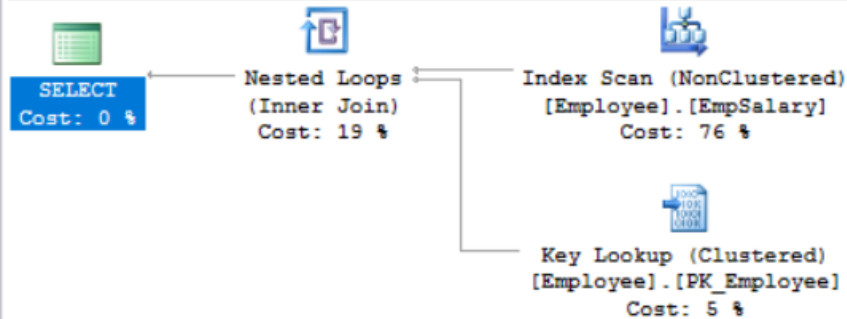
```
SELECT EmployeeID, FirstName, LastName  
FROM Employee  
WHERE Salary > 100000/1.10;
```


Tips & tricks:

(3) avoid calculations, isolate columns

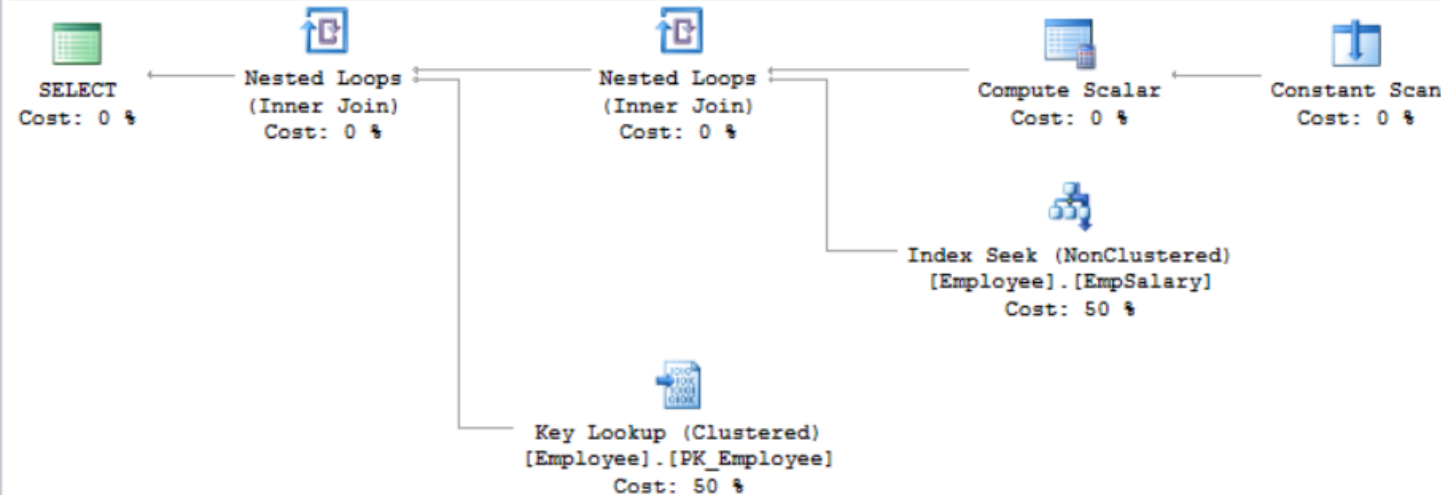
Query 1: Query cost (relative to the batch): 91%

```
SELECT [EmployeeID],[FirstName],[LastName] FROM [Employee] WHERE [Salary]*@1>@2
```



Query 2: Query cost (relative to the batch): 9%

```
SELECT [EmployeeID],[FirstName],[LastName] FROM [Employee] WHERE [Salary]>@1/@2
```



Key lookup:

The non-clustered index EmpSalary, holds in each leaf a reference to the location of the total record in the clustered index. Following this reference is called “key lookup”.

Tips & tricks

(4) prefer OUTER JOIN above UNION

-- BAD

```
SELECT lastname,firstname,orderid
from Employee e join Orders o on e.EmployeeID =
o.employeeid
union
select lastname,firstname,null
from Employee where EmployeeID not in (select EmployeeID
from Orders)
```

-- GOOD

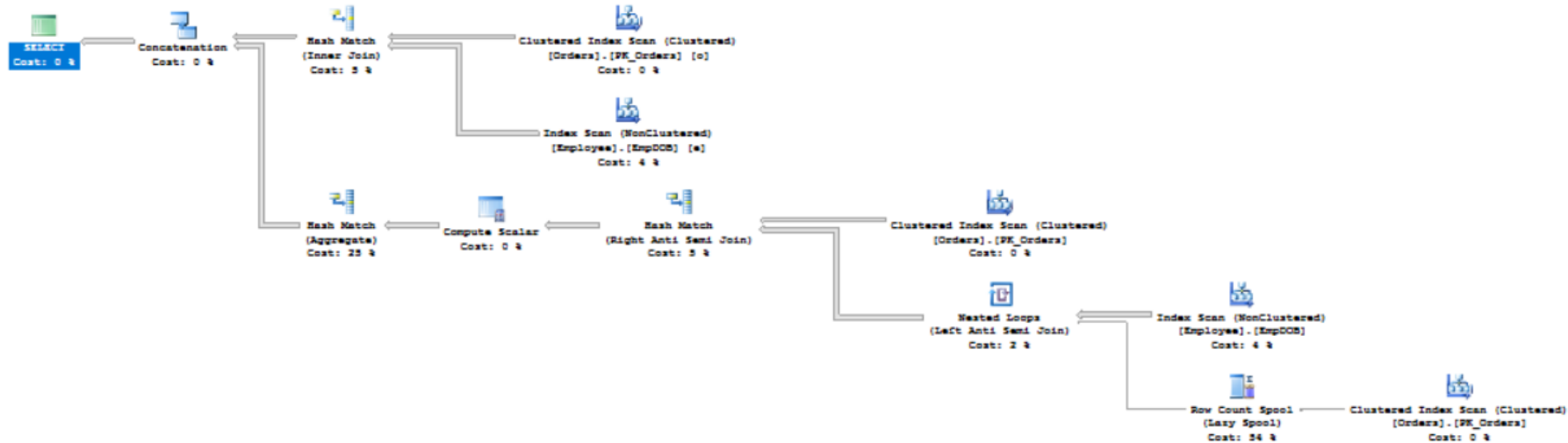
```
SELECT lastname,firstname,orderid
from Employee e left join Orders o on e.EmployeeID =
o.employeeid;
```

Tips & tricks:

(4) prefer OUTER JOIN above UNION

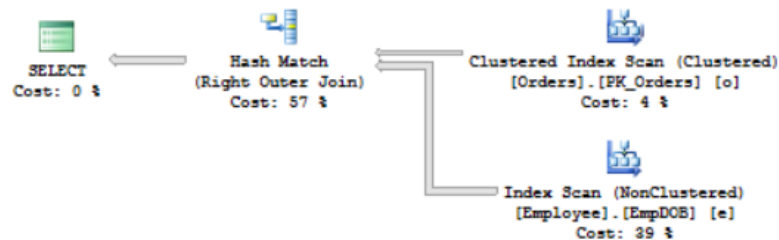
Query 1: Query cost (relative to the batch): 91%

SELECT lastname,firstname,orderid from Employee e join Orders o on e.EmployeeID = o.employeeid union select lastname,firstname,null from



Query 2: Query cost (relative to the batch): 9%

SELECT lastname,firstname,orderid from Employee e left join Orders o on e.EmployeeID = o.employeeid



Tips & tricks

(5) avoid ANY and ALL

-- BAD

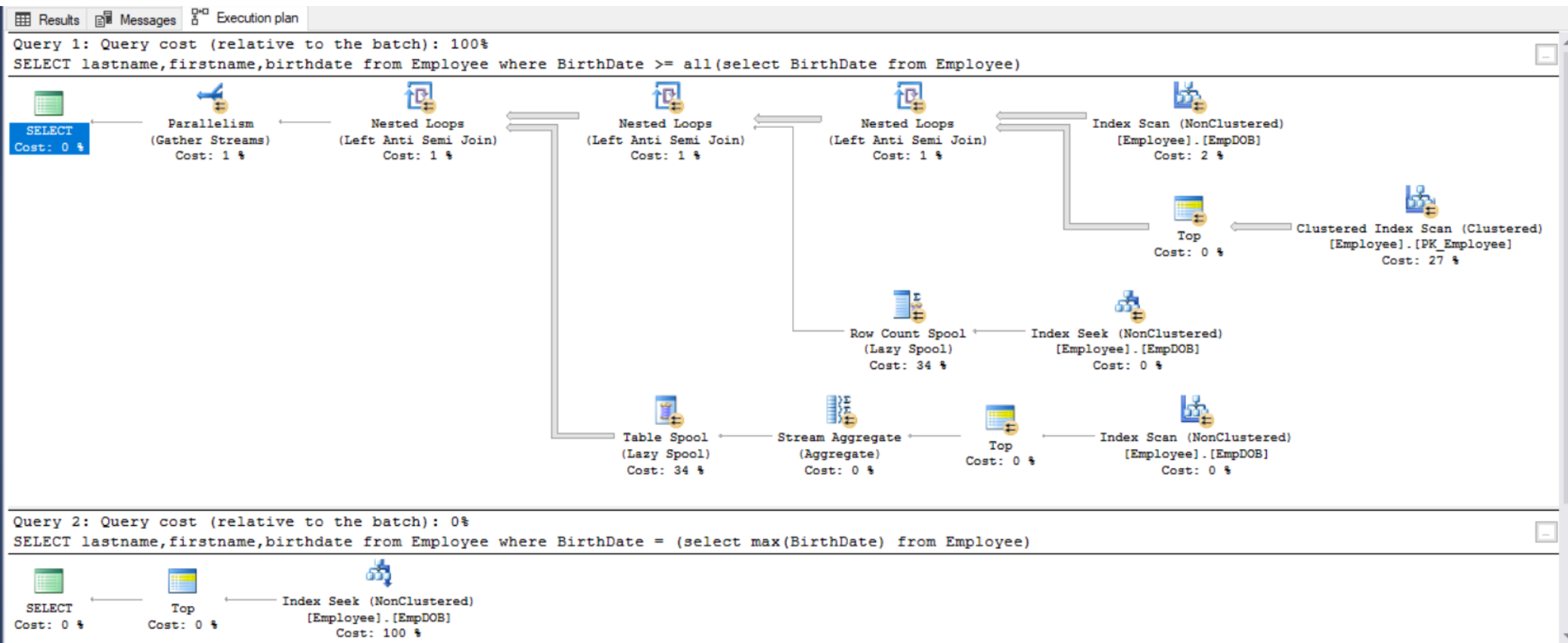
```
SELECT lastname,firstname,birthdate  
from Employee where BirthDate >=  
all(select BirthDate from Employee)
```

-- GOOD

```
SELECT lastname,firstname,birthdate  
from Employee where BirthDate =  
(select max(BirthDate) from Employee)
```

Tips & tricks:

(5) avoid ANY and ALL



Quiz 1/5

Is the following index a good fit for the query?

```
CREATE INDEX tbl_idx ON tbl (date_column);
```

```
SELECT * FROM tbl
```

```
WHERE YEAR(date_column) = 2017;
```

- A. Good fit: No need to change anything
- B. Bad fit: Changing the index or query could improve performance

Quiz 2/5

Is the following index a good fit for the query?

```
CREATE INDEX tbl_idx ON tbl (a, date_column);
```

```
SELECT TOP 1 * FROM tbl  
  WHERE a = 12  
  ORDER BY date_column DESC;
```

- A. Good fit: No need to change anything
- B. Bad fit: Changing the index or query could improve performance

Quiz 3/5

Is the following index a good fit for both queries?

```
CREATE INDEX tbl_idx ON tbl (a, b);
```

```
SELECT * FROM tbl  
WHERE a = 123 AND b = 1;
```

```
SELECT * FROM tbl WHERE b = 123;
```

- A. Good fit: No need to change anything
- B. Bad fit: Changing the index or query could improve performance

Quiz 4/5

Is the following index a good fit for the query?

```
CREATE INDEX tbl_idx ON tbl (text);
```

```
SELECT *
```

```
FROM tbl
```

```
WHERE text LIKE 'TJ%';
```

- A. Good fit: No need to change anything
- B. Bad fit: Changing the index or query could improve performance

Quiz 5/5

This question is different. First consider the following index and query:

```
CREATE INDEX tbl_idx ON tbl (a, date_column);
```

```
SELECT date_column, count(*)  
  FROM tbl  
 WHERE a = 123  
 GROUP BY date_column;
```

Let's say this query returns at least a few rows.
To implement a new functional requirement, another condition ($b = 1$) is added to the where clause:

```
SELECT date_column, count(*) FROM tbl  
 WHERE a = 123 AND b = 1  
 GROUP BY date_column;
```

How will the change affect performance:

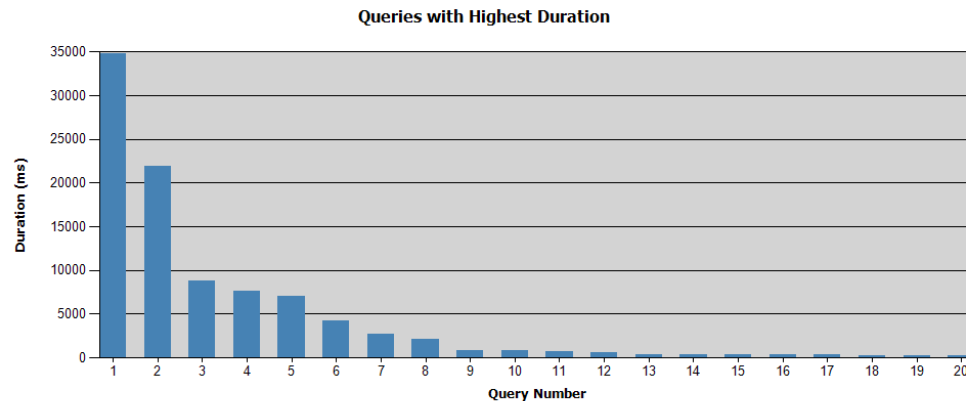
- A. Same: Query performance stays about the same
- B. Not enough information: Definite answer cannot be given
- C. Slower: Query takes more time
- D. Faster: Query take less time

Looking for "expensive" queries

Install the performance dashboard according to the guidelines in <http://blog.sqlauthority.com/2014/09/22/sql-server-ssms-performance-dashboard-installation-and-configuration/>

Start via Database/Reports/Custom reports...

example of a rapport: Expensive queries by duration:



Query Number	Representative Query	Total Executes	Cached Query Count	Unique Plan Count	Highest Plan Generation	Earliest Plan Cached	Cumulative CPU (ms)					Cumulative Physical Reads	Cumulative Logical Reads
							Total	Total	Max	Avg	Min	Total	Total
1	select * from person.person order by person.person.LastName,firstName	6	3	1	1	3/4/2015 2:22:41 PM	4,378.468	34,813.075	24,848.507	5,802.179	1,874.773	3,926	415,904
2	SELECT [NULL](case dmi.mirroring_redo_queue_type when NVUNLIMITED then 0 else dmi.mirroring_redo_queue end),0) AS [MirroringRedoQueueMaxSize], [NULL](dmi.mirroring_connection_timeout,0) AS [MirroringTimeout], [NULL](dmi.mirroring_partner_name,') AS [MirroringPartner], [NULL](dmi.mirroring_partner_instance,') AS [MirroringPartnerInstance], [NULL](dmi.mirroring_role,0) AS [MirroringRole], [NULL](dmi.mirroring_safety_level + 1, 0) AS [MirroringSafetyLevel], [NULL](dmi.mirroring_state + 1, 0) AS [MirroringStatus], [NULL](dmi.mirroring_witness_name,') AS [MirroringWitness]	419	1	1	1	3/4/2015 11:51:51 AM	6,302.242	21,859.012	346.418	52.169	0.489	6,199	56,774